

作业四

15.4-2

不利用b、只利用c来重构 LCS

- **初始化：** i 和 j 分别初始化为序列 X 和 Y 的长度，表示从表的右下角开始追溯。创建一个空列表 LCS 用于存储公共子序列的字符。
- **匹配字符：** 如果 $X[i-1] = Y[j-1]$ ，说明这两个字符属于 LCS，将其加入 LCS 并向左上移动。**优先方向：** 如果字符不匹配，比较表 c 中的值：如果 $c[i-1][j] \geq c[i][j-1]$ ，说明向上移动能获得更大的子序列长度；否则，向左移动。
- 最后返回完整的 LCS 作为最终结果。
- 对于一个完整的c表来说，重构进行移动的总移动次数不超过行m和列n的总和，即 $m+n$ ，则时间复杂度为 $O(m + n)$

输入：已完成的 LCS 长度表 c ，序列 $X = \langle x_1, x_2, \dots, x_m \rangle$ ，序列 $Y = \langle y_1, y_2, \dots, y_n \rangle$

输出：序列 X 和 Y 的最长公共子序列 LCS

```
1. i ← length(X) // 初始化 i 为序列 X 的长度
2. j ← length(Y) // 初始化 j 为序列 Y 的长度
3. LCS ← new Array // 用于存储重构出的 LCS

4. while i > 0 and j > 0 do
5.     if X[i] = Y[j] then // 如果当前字符匹配
6.         将 X[i] 添加到 LCS
7.         i ← i - 1 // 向左上移动
8.         j ← j - 1
9.     else if c[i-1][j] ≥ c[i][j-1] then // 如果向上移动的值较大
10.        i ← i - 1 // 向上移动
11.    else // 向左移动的值较大
12.        j ← j - 1 // 向左移动
13.    end if
14. end while

15. 反转 LCS // 因为是从后向前追溯
16. 返回 LCS
```

15.4-3

带备忘优化的 **LCS-LENGTH** 算法

输入：字符串 $X[1..m]$ 和 $Y[1..n]$

输出：最长公共子序列（LCS）的长度

1. 初始化一个备忘录表 $\text{memo}[0..m][0..n]$ ，所有元素初始值为未定义（未计算）。
2. 定义递归函数 $\text{LCS}(i, j)$ ：
3. 如果 $i = 0$ 或 $j = 0$ ：
 - 4. 返回 0 // 基础情况：任意一个字符串为空时，LCS 长度为 0。
 - 5. 如果 $\text{memo}[i][j]$ 已定义：
 - 6. 返回 $\text{memo}[i][j]$ // 返回缓存的结果。
 - 7. 如果 $X[i] = Y[j]$ ：
 - 8. $\text{memo}[i][j] \leftarrow 1 + \text{LCS}(i-1, j-1)$ // 当前字符匹配，将其加入 LCS。
 - 9. 否则：
 - 10. $\text{memo}[i][j] \leftarrow \max(\text{LCS}(i-1, j), \text{LCS}(i, j-1))$ // 当前字符不匹配，取两种可能的最大值。
 - 11. 返回 $\text{memo}[i][j]$ 。
 - 12. 调用 $\text{LCS}(m, n)$ 来计算完整字符串 X 和 Y 的 LCS 长度。
 - 13. 返回 $\text{memo}[m][n]$ 中存储的结果。

1. 当 $i=0$ 或 $j=0$ 时，表示字符串 X 或 Y 为空，此时 LCS 长度为 0。
2. 如果 $X[i]=Y[j]$ ，那么 $X[i]$ 和 $Y[j]$ 属于 LCS，当前 LCS 长度为 $1+\text{LCS}(i-1,j-1)$ 。
如果 $X[i]\neq Y[j]$ ，则需要在以下两种情况中选择较大者：
排除 $X[i]$ ：即 $\text{LCS}(i-1,j)$ ；排除 $Y[j]$ ：即 $\text{LCS}(i,j-1)$ 。
3. 备忘： $\text{memo}[i][j]$ 用于存储子问题 (i,j) 的计算结果。如果该结果已计算过，则直接返回，避免重复计算。
4. 时间复杂度：每个子问题 (i,j) 最多计算一次，总共有 $m \times n$ 个子问题，故时间复杂度为 $O(mn)$ 。
5. 空间复杂度：需要一个 $O(mn)$ 的备忘录表，以及递归调用的栈空间（最多 $O(m+n)$ ）。

15.4-5

定义一个数组 dp ，其中 $\text{dp}[i]$ 表示以索引 i 结尾的最长递增子序列的长度。同时，保持一个额外的数组来记录每个位置的前驱元素，以返回 **实际的最长单调递增子序列**，以便从后向前回溯实际的子序列。

1. 初始化 dp 数组，其中每个元素初始值为 1，表示每个元素至少自己可以形成一个长度为 1 的子序列。
2. 初始化一个 prev 数组，用来记录每个元素的前驱元素，初始时所有元素都没有前驱（设置为 -1）。

3. 对于每个 i (从 1 到 $n-1$)，遍历 j (从 0 到 $i-1$)，如果 $A[j] < A[i]$ ，则更新 $dp[i]$ 并记录前驱。
4. 找到 dp 数组中的最大值，并从 $prev$ 数组回溯得到最长递增子序列。

伪代码

```
Input: 数组 A[0..n-1]
Output: 最长单调递增子序列的长度和实际子序列
```

1. 初始化 dp 数组， $dp[i] = 1$ 对于所有 $i \in [0, n-1]$
2. 初始化 $prev$ 数组， $prev[i] = -1$ 对于所有 $i \in [0, n-1]$
3. 对于 $i = 1$ 到 $n-1$ ：
 - a. 对于 $j = 0$ 到 $i-1$ ：
 - i. 如果 $A[j] < A[i]$ ：
更新 $dp[i] = \max(dp[i], dp[j] + 1)$
如果 $dp[i] == dp[j] + 1$, 更新 $prev[i] = j$
4. 找到 dp 数组中的最大值 max_len 和对应的索引 idx
5. 回溯 $prev$ 数组，得到最长单调递增子序列
6. 返回 max_len 和子序列

时间复杂度分析

- 外层和内层循环的时间复杂度为 $O(n^2)$ ，因此总时间复杂度是 $O(n^2)$ 。
- 回溯最长递增子序列的部分时间复杂度为 $O(n)$ ，因此总时间复杂度仍然是 $O(n^2)$ 。